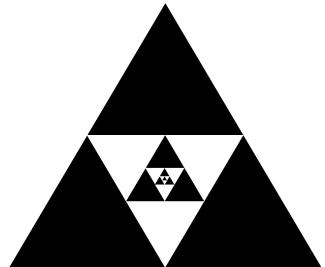
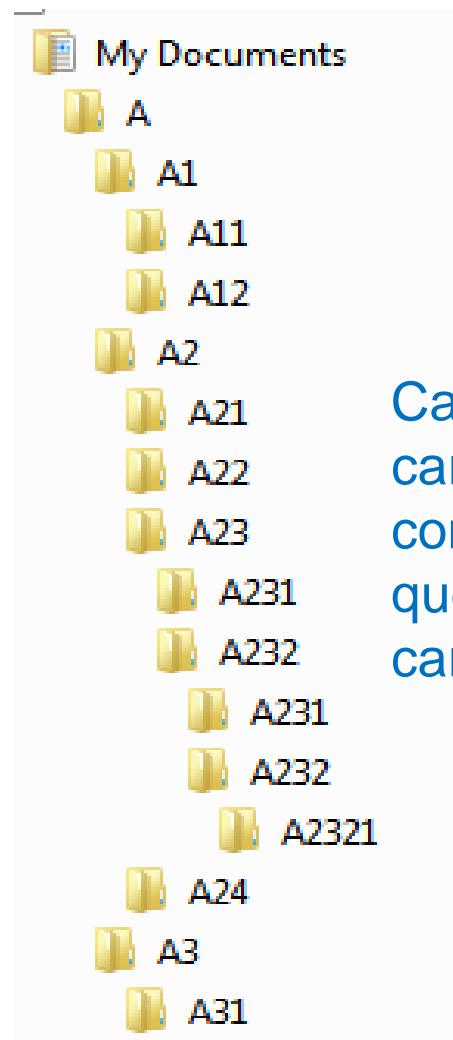
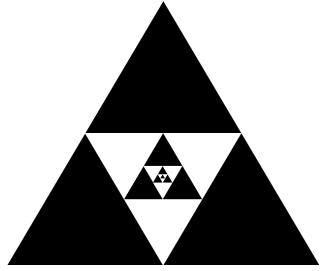




Triángulo dentro de
triángulo dentro de
triángulo dentro
de triángulo

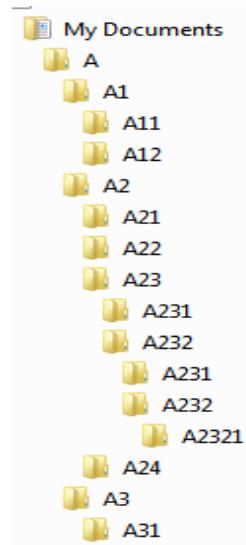
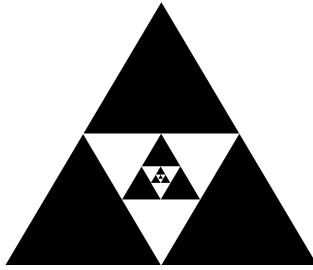


Matriozka dentro de
matriozka dentro de
matrioska
dentro de matriozka



Carpeta que contienen carpetas que contienen carpetas que contienen carpetas





Recursividad - Microsoft Visual Studio

File Edit View Refactor Project Build Debug Team Data Tools Architecture Test Analyze Window Help

Debug cadena

Code F7
Solution Explorer Ctrl+W, S
Team Explorer Ctrl+W, M
Server Explorer Ctrl+W, L
Architecture Explorer Ctrl+W, N
Call Hierarchy Ctrl+W, K
Class View Ctrl+W, C
Code Definition Window Ctrl+W, D
Object Browser Ctrl+W, J
Error List Ctrl+W, E
Output Ctrl+W, O
Start Page
Task List Ctrl+W, T
Toolbox Ctrl+W, X
Find Results
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
Navigate Backward Ctrl+-
Navigate Forward Ctrl+Shift+-
Next Task
Previous Task
Properties Window Ctrl+W, P
Property Pages Shift+F4
Prueba de Hanoi
Prueba de Búsqueda Binaria
Prueba de Fibonacci

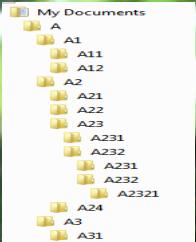
Command Window Ctrl+W, A
Web Browser Ctrl+W, W
Layer Explorer
Macro Explorer Alt+F8
Source Control Explorer
UML Model Explorer Ctrl+\, Ctrl+U
Bookmark Window Ctrl+W, B
Android Device Logging
Document Outline Ctrl+W, U
History
Pending Changes
Property Manager
Resource View Ctrl+W, R
F# Interactive Ctrl+Alt+F
Performance Explorer
Code Metrics Results



Menú que contiene
menú que contiene
menú que contiene
menú ... que contiene
menú ...



Recursividad



Recursión (Recursividad)

Base para una estrategia de solución de problemas

Resolver un problema complejo reduciéndolo o diviéndolo a uno o más **subproblemas**

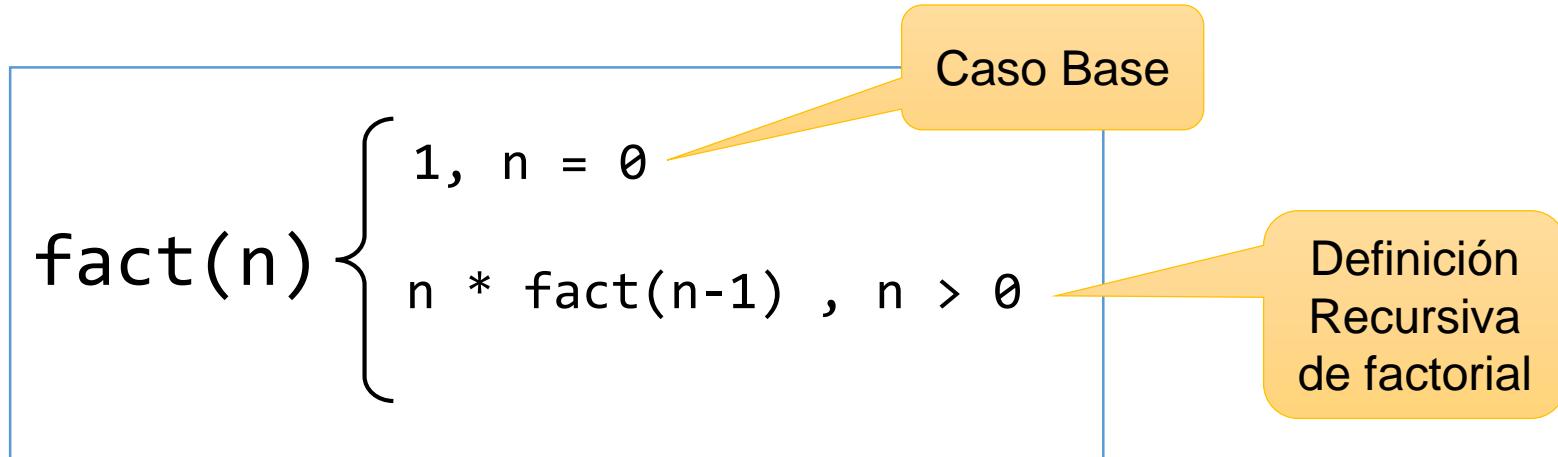
- que tienen “**la misma estructura**” que el problema original
- pero que son “**más simples**” de resolver que el problema original

En cada paso el subproblema **se divide**, usando un mismo procedimiento, en subproblemas **más simples**

Los subproblemas llegarán a ser tan simples que **no hará falta seguir dividiéndolos** para resolverlos

Para resolver el problema original puede bastar con la solución de uno de los subproblemas o puede que haya que **combinar** las soluciones de cada subproblema

factorial



$$\text{fact}(5) = 1 * 2 * 3 * 4 * 5 = 120$$

$$\text{fact}(5) = 5 * \text{fact}(4)$$

$$5 * 4 * 3 * 2 * 1 * 1$$

120

$$4 * \text{fact}(3)$$

$$4 * 3 * 2 * 1 * 1$$

$$3 * \text{fact}(2)$$

$$3 * 2 * 1 * 1$$

$$2 * \text{fact}(1)$$

$$2 * 1 * 1$$

$$1 * \text{fact}(0)$$

$$1 * 1$$

División en subproblemas

Solución del problema original

Combinar soluciones de los subproblemas

factorial

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:
```

Caso Base. El problema es tan simple que se resuelve directamente

Se reduce a un problema más simple

Se combina la solución del subproblema

Subproblema. Se llama al método dentro de misma definición del método

Estructura general de un algoritmo recursivo

Algoritmo RECURSIVO

IF (Problema Simple)

 Resolverlo directamente

Condición
de Parada

Caso Base

ELSE

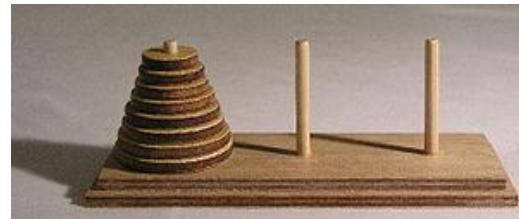
 Dividir en subproblemas P1, P2, ..., Pn

Resolver(P1); Resolver(P2); ... Resolver(Pn)

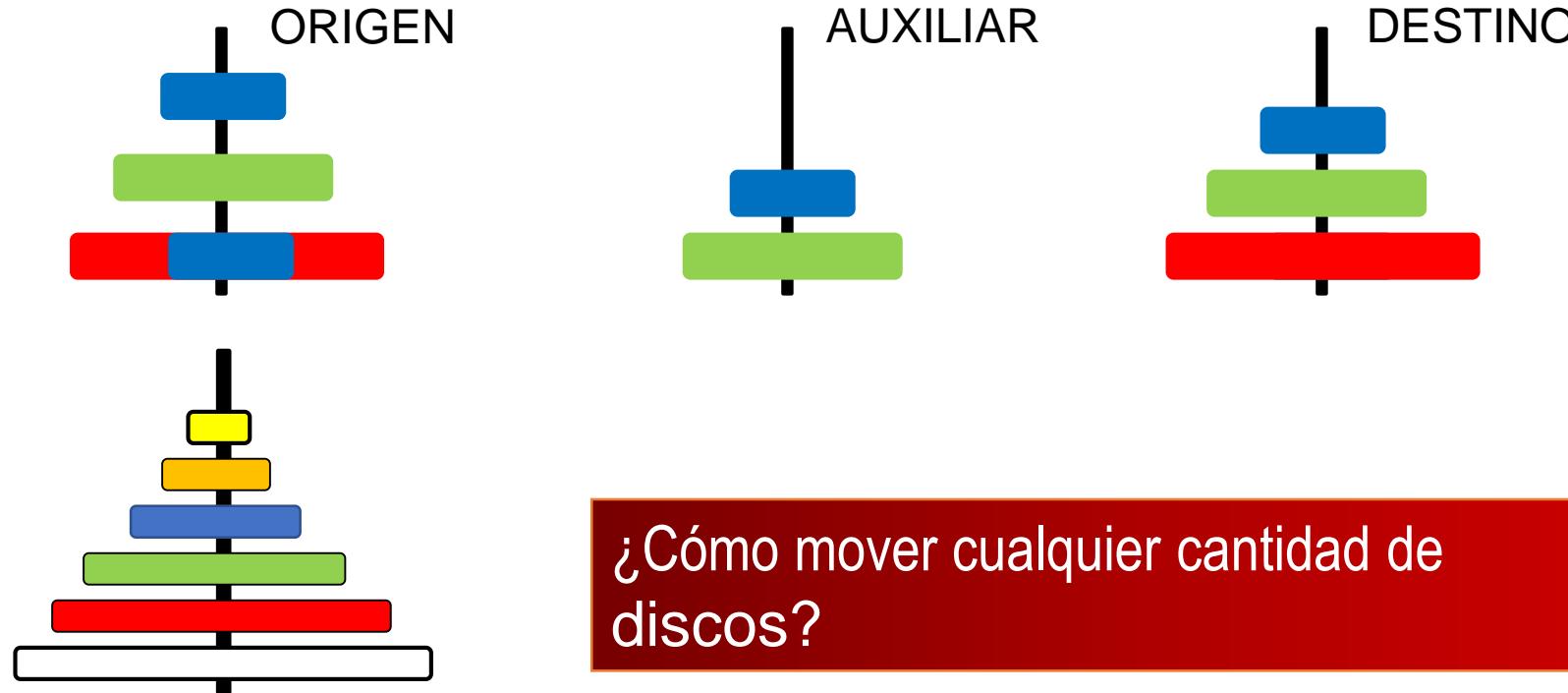
 Combinar las soluciones de cada subproblema

Resolver de
manera
recursiva

Torres de Hanoi



Tenemos en una torre una pila de discos de mayor a menor y queremos pasarlos a la tercera torre usando la del medio como auxiliar, pero solo se puede mover un disco a la vez y nunca se puede poner un disco de mayor tamaño sobre uno menor tamaño

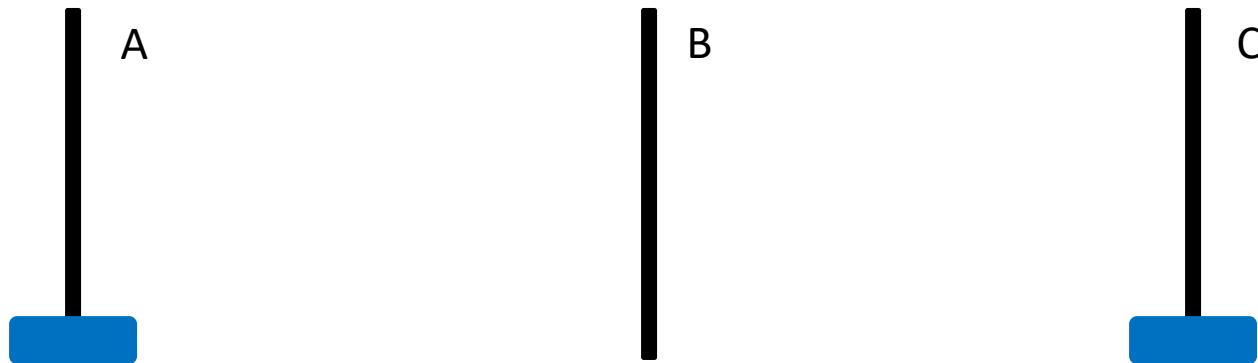


¿Cómo mover cualquier cantidad de discos?

Torres de Hanoi

CASO BASE

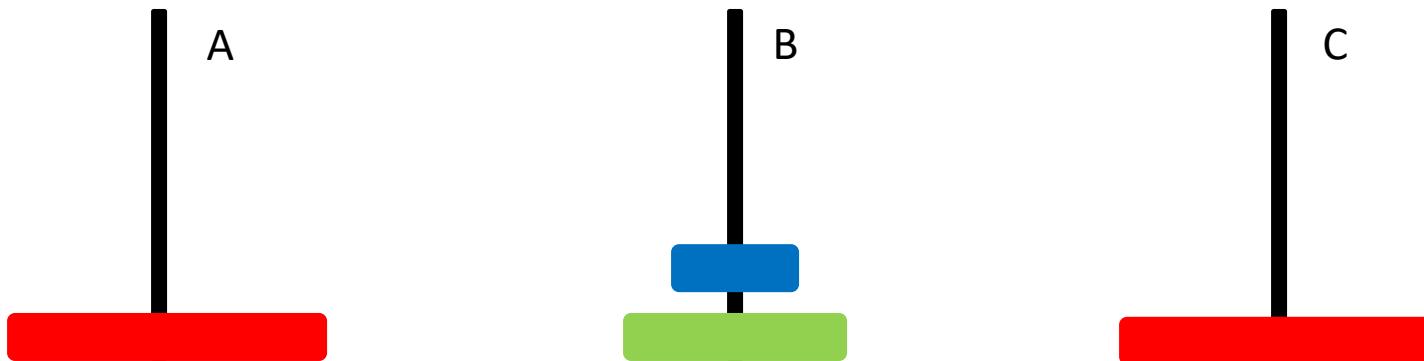
Si la cantidad de discos a mover es 1 pues lo movemos



Torres de Hanoi

CASO BASE

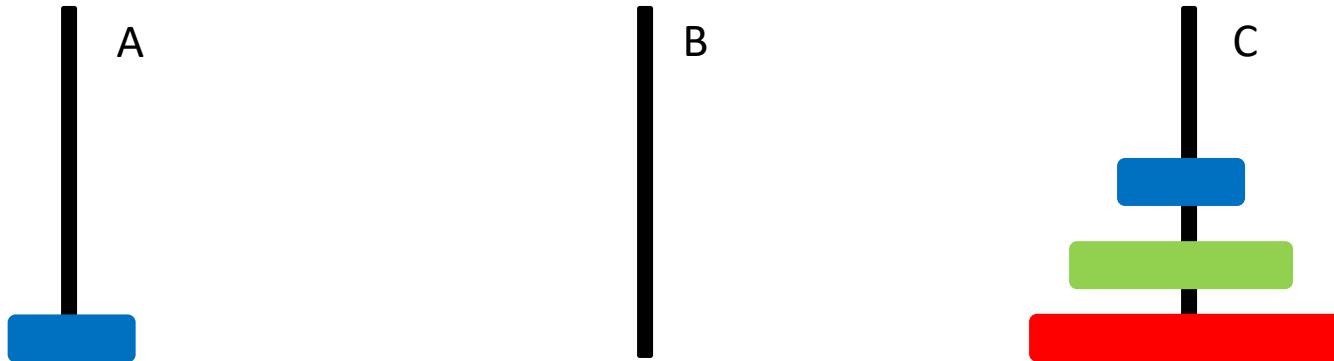
Mover 1 disco de un ORIGEN A a un DESTINO C



Torres de Hanoi

CASO BASE

Mover 1 disco de un ORIGEN A a un DESTINO C



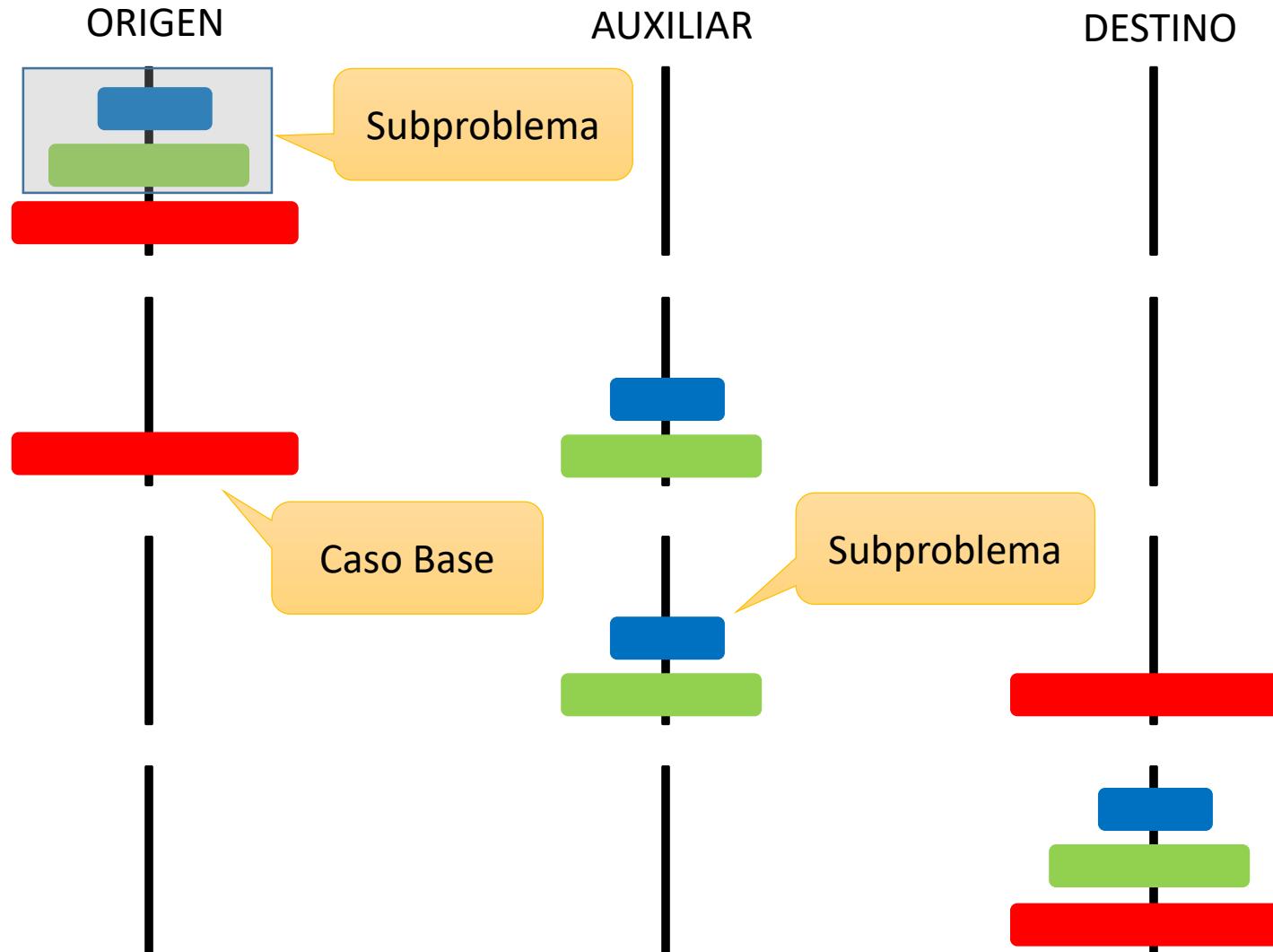
Torres de Hanoi

SUBPROBLEMA

Si la cantidad de discos es mayor que 1

1. Mover ($n-1$) discos de ORIGEN a AUXILIAR
2. Mover el disco que queda en ORIGEN para DESTINO
3. Mover los ($n-1$) de AUXILIAR a DESTINO

Torres de Hanoi



Torres de Hanoi

Caso Base de la recursión

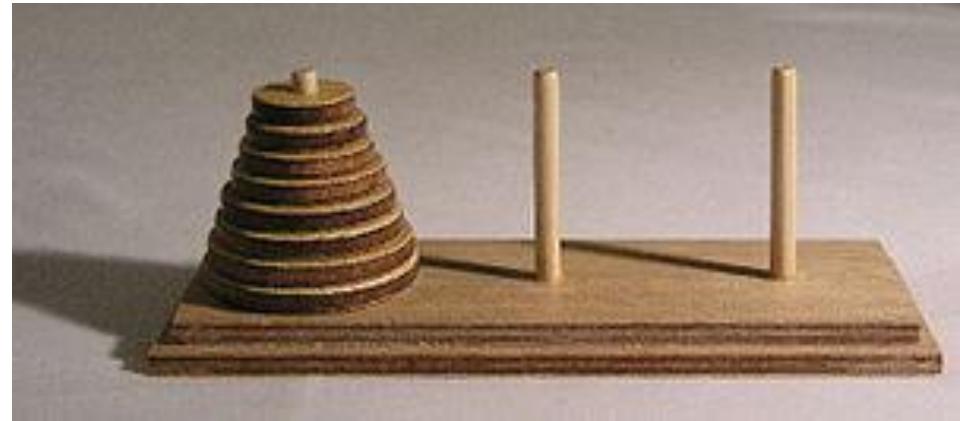
```
def Hanoi(n, origen, auxiliar, destino): Converge al caso base
    if n == 1:
        print(f'Mueve de {origen} a {destino}')
    else:
        Hanoi(n-1, origen, destino, auxiliar)
        print(f'Mueve de {origen} a {destino}')
        Hanoi(n-1, auxiliar, origen, destino)

Hanoi(4, "A", "B", "C")
```

Converge al caso base

Intente programarlo sin recursividad

DEMO



Ejercicios

1. Escriba una función recursiva `invierte(lista)` para que devuelva una lista con los mismos valores invertidos. Es decir que
`invierte([10, 4, 20, 11])`
debe devolver la lista `[11, 20, 4, 10]`
2. Defina una función iterativa y una recursiva para calcular el elemento enésimo de la sucesión de Fibonacci. Recuerde que la sucesión de Fibonacci es `1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
Analice la diferencia de tiempo de ambas.
3. La expresión Python `a**n` calcula el valor de `a` elevado a la `n`. Suponga que no disponemos del operador `**`. Escriba una función `potencia(a,n)` que devuelva el valor de `a` elevado a la potencia `n`
4. Un número es **superprimo** si es primo y si al quitarle el último dígito sigue siendo superprimo. Por ejemplo 71 es superprimo porque es primo y además 7 es superprimo. Implemente una función `superprimo`

Ejercicios

5. *Recuerde la función que hace una **búsqueda binaria** para encontrar la posición de un elemento en una lista ordenada. El algoritmo consistía en ir dividiendo la lista en mitades e ir buscando en cuál de las dos mitades podría estar el valor buscado hasta que el valor buscado esté en el medio de la lista o que esta ya no pudiera seguirse dividiendo. Implemente una versión recursiva de esta función. Implemente una tal función `busquedaBinaria(x, lista)` que devuelva la posición en que está `x` en `lista`